Master of Puppets: Cooperative Multitasking for In Situ Processing

Dmitriy Morozov* Zarija Lukić†
Lawrence Berkeley National Laboratory

June 12, 2016

Abstract

Modern scientific and engineering simulations track the time evolution of billions of elements. For such large runs, storing most time steps for later analysis is not a viable strategy. It is far more efficient to analyze the simulation data while it is still in memory. In this paper, we present a novel design for running multiple codes in situ: using coroutines and position-independent executables we enable cooperative multitasking between simulation and analysis, allowing the same executables to post-process simulation output, as well as to process it on the fly, both in situ and in transit. We present Henson,[1] an implementation of our design, and illustrate its versatility by tackling analysis tasks with different computational requirements. Our design differs significantly from the existing frameworks and offers an efficient and robust approach to integrating multiple codes on modern supercomputers. The presented techniques can also be integrated into other in situ frameworks.

1 Introduction

Many scientific fields rely on simulations to produce models and predictions. Typical examples are astrophysics and cosmology, climate studies, plasma physics, and neural simulations, where the number of computational elements, for example, dark matter particles or neural synapses, reaches into the trillions. A single time snapshot from such runs is tens of terabytes. In this regime, the traditional approach of running a CPU-intensive simulation, saving many outputs to disk, and analyzing them later is hopelessly inefficient.

To make matters worse, while there are several HPC centers around the world that allow scientists to obtain time allocations large enough to produce such simulations, there is virtually no way to apply for petabytes of disk storage for their output. Therefore, a significant challenge for computational science is how to efficiently run simulation and analysis codes *in situ*, without saving (most of) the data to disk. This problem is the main motivation of our work.

Even if there was disk space, it would be impractical to save a large number of time steps — the slowdown due to I/O would be high. At the same time, the data could be analyzed while it's still resident in memory: many forms of analysis produce output orders of magnitude smaller than the input — halo catalogs or light cones in cosmological simulations; changes in average temperatures over time in climate simulations; cumulative energy distributions for particles in plasma simulations — making it both possible and sensible to save the results of the analysis, while either abandoning the simulation data completely, or storing only a few snapshots.

It may seem that the supercomputers' restrictions and the desire to share memory force simulation and analysis codes to be tightly coupled by being compiled into a single executable. In this paper we show that tight coupling is not required. The codes can remain separate yet execute on the same nodes. Crucially, they can share the same memory and data without any changes to their memory management facilities.

*dmitriy@mrzv.org
†zarija@lbl.gov
[1] https://github.com/mrzv/henson

2 Cooperative Multitasking

Our solution depends on two ingredients: position-independent executables and coroutines.

Position-independent executables. To let multiple executables share memory, as well as to avoid limitations of some supercomputers, such as the absence of `fork` on IBM BG/Q systems, we can compile our codes as *position-independent executables*. The resulting binaries are both executables and dynamic libraries.

If one compiles an analysis code as a position-independent executable, one can launch it as a standalone process to analyze a snapshot of a simulation saved on disk. But it also becomes possible to load the code inside a different process, using the `dlopen/dlsym` facilities of `libdl` to get the address of its `main`. Listing 1 presents the relevant code snippet.

```
typedef  int  (*MainType)(int argc, char *argv[]);
void*    lib      = dlopen(fn.c_str(), RTLD_LAZY);
MainType lib_main = (MainType) dlsym(lib, "main");
```

Listing 1: `dlopen/dlsym` example.

Loading simulation and analysis executables inside one process has a significant advantage. All the routines share the same address space — there is no process isolation enforced by the operating system. Therefore, they can exchange data by simply passing pointers to each other, without any changes to their memory management facilities. Achieving such zero-copy regime with executables running as separate processes is considerably more complicated.

Coroutines. Once the `main` functions are in memory, we need to switch control between them. It is possible for the simulation to call the analysis code directly, as a subroutine. But this puts a restriction on the analysis, which wouldn't exist if we used separate processes: when returning back to the simulation, the analysis would inevitably lose its state, both the stack and the program counter.

What we face is a classical problem with subroutines: they have a single entry point, and, when returning, they lose their stack frame. This problem has an elegant solution. *Coroutines*, a generalization of subroutines, described in detail in the first volume of Knuth's monograph [1, Section 1.4.2], maintain their state across invocations. Switching from one coroutine to another changes the execution context, including the stack pointer, program counter, registers, etc. As a consequence, not only is the coroutine's data preserved, but when we switch back to it, the execution resumes exactly where it left off.

There are C++ libraries that provide the coroutine functionality. By default, Henson uses `libcoro`,[2] included in its distribution, which allows us to create an independent context for each executable. The switches between contexts happen *cooperatively*, when either simulation or analysis explicitly returns control via a `yield` function, described in more detail in the next section.

3 Henson

We now describe a system, Henson, built on the principles of the previous section. Henson consists of three parts: the application that controls the execution flow between the different codes, referred to as *puppets* inside the application; a small C library that the puppets must link (the library wraps `libcoro` and provides functions to control execution and data exchange); and auxiliary tools (as well as examples) to simplify transition from in situ to in transit analysis regimes.

Controller. `henson`, the application, controls execution. The user specifies what codes to run and how to run them in a script. A sample script appears in Listing 4. The first lines define puppets used in the code. The text after the equals sign is interpreted as a command line: the first word is the executable, while the rest of the symbols are parsed and passed to the `main` function. Once the puppets are loaded, the script specifies how to alternate execution between them.

[2] `http://software.schmorp.de/pkg/libcoro.html`

Puppets. Listing 2 gives an example of a puppet; it illustrates a typical time step of a simulation. After the computation is finished, the simulation exposes its data to other puppets by calling `henson_save_*`, and yields control back to `henson` by calling `henson_yield`. These functions are provided by `libhenson`, the C library that all the puppets must link.

```
while (/* main time loop */)
{
  // ... (simulation time step)
  henson_save_double("redshift", z);
  henson_save_array("x", &P[0].Pos[0], sizeof(float), count, sizeof(struct particle_data));
  henson_save_array("y", &P[0].Pos[1], sizeof(float), count, sizeof(struct particle_data));
  henson_save_array("z", &P[0].Pos[2], sizeof(float), count, sizeof(struct particle_data));
  henson_yield();
}
```

Listing 2: Time step in a cosmological simulation puppet. Particle quantities are stored in `struct particle_data`. Our analysis uses only positions; we set element size to `sizeof(float)`, while using the size of `particle_data` as the stride. `count` is the number of particles on the local processor.

Henson provides a shared table that maps strings to values. When a puppet saves an array, it does not actually copy any elements, but rather stores the pointer and array metadata in the table. An array stores the address of its first element, the size of individual elements, their number, and the stride between elements; see Listing 2 for an example.

Having read this metadata, analysis code can access the respective arrays directly. Listing 3 shows a skeleton of the analysis code. Analysis itself may choose to yield execution to `henson`, for example, if its output may need to be processed by another code.

```
double redshift; size_t count, dtype, stride;
float *x, *y, *z;
henson_load_double("redshift", &redshift);
henson_load_array("x", &x, &dtype, &count, &stride);
henson_load_array("y", &y, &dtype, &count, &stride);
henson_load_array("z", &z, &dtype, &count, &stride);
// analyze the data
henson_save_pointer("analysis-data", analysis_data);
henson_yield();
```

Listing 3: Analysis skeleton.

Execution groups. Henson supports multiple execution groups, each specified as a separate `while`-loop in the script. Listing 4 defines two execution groups, `producer` and `consumer`. When launching `henson`, a user may specify on the command line how many processors to allocate to each execution group.

Henson supports multiple execution groups to allow in transit analysis, where data moves from one group to another, running on a separate set of processors. To identify those processors, group names are helpful inside puppets: a puppet may request an MPI inter-communicator that connects the processors of its local execution group with those of a remote group by calling `henson_get_intercomm(remote_group_name)`. In Listing 4, group names (`producer` and `consumer`) are passed to `send` and `receive` puppets precisely for this reason.

MPI. Henson initializes MPI, so puppets should not repeat the initialization individually. Similarly, each puppet should restrict its communication to an MPI intra-communicator within its execution group. The latter can be obtained with an explicit call to `henson_get_world`, but `libhenson` also includes MPI wrappers (built on top of the PMPI interface). These wrappers, when running under Henson, disable MPI initialization and transparently replace `MPI_COMM_WORLD` by the result of `henson_get_world` in all operations. These wrappers allow the same codes to run independently or under Henson, both with a single or multiple execution groups, with no changes.

```
sim         = ../../L-Gadget3/L-Gadget3 $gadget_settings(gadget-defaultx256)
tess        = ./tess -w _ _ _ _ _ 0. 0. 0. $max_x(100.) $max_y(100.) $max_z(100.)
entropy     = ./entropy _ $entropy_out(entropy-256.txt)
lightcone   = ./lightcone 1 1 1 0.524 lightcone.out

send    = ../henson/tools/send    --async consumer x,y,z:array t:size_t redshift:double
receive = ../henson/tools/receive --async producer x,y,z:array t:size_t redshift:double

producer while sim:
    sim
    lightcone
    send

consumer while receive:
    receive
    *tess
    entropy
```

Listing 4: Henson script to (asynchronously) exchange data between execution groups, `producer` and `consumer`. The two `while`-loops execute in parallel; `send` and `receive` puppets move data between the processors on demand.

Data flow. Individual puppets don't have to be simulation or analysis codes, they can also be routines that exchange data between execution groups. Henson includes two tools, `send` and `receive`, which not only serve as examples of such puppets, but are also useful for real applications. `send` looks up the names in the shared table and sends their contents to the remote group; `receive` receives the remote data and inserts them into the local table. Both tools can communicate between execution groups with different numbers of processors.

The two routines can be run in synchronous or asynchronous mode; `--async` flag in Listing 4 indicates the latter. In synchronous mode, the two routines wait for each other, so each time step of the simulation is sent over to the analysis. In the asynchronous mode, `send` checks if `receive` has requested more data, in which case it fulfills the request. When the temporal resolution of the simulation is finer than what the analysis needs, the asynchronous mode has the obvious advantage: simulation and analysis proceed at their own pace, pausing to exchange data only when necessary.

The advantage of our design is that simulation and analysis don't require any changes to accommodate different execution regimes: simulation posts pointers to data in a shared table, analysis reads them from the table, but each one is oblivious to what happens when they are not running. This allows other puppets (like `send` and `receive` above) to move data between different nodes, enabling the in transit analysis, illustrated in Listing 4. At the same time, the simulation and analysis codes require no changes to run in situ; the user simply modifies the script to replace two execution groups by one, as in Listing 5.

```
world while sim:
    sim
    lightcone
    *tess
    entropy
```

Listing 5: In situ version of the script in Listing 4, with identical puppet definitions omitted.

4 Experiments

To evaluate Henson, we conducted a series of experiments at the National Energy Research Scientific Computing Center (NERSC) on Edison, a Cray XC30 with 5,576 nodes, each with 24 cores (Intel Ivy Bridge 2.4 GHz) and 64 GB RAM.

We ran several N-body simulations using Gadget [2], a widely used code for modelling the evolution of cosmic structure, which we modified to save in Henson's data table, after every time step, several internal variables, including particle positions and redshift, and to call `henson_yield`. Listing 2 shows all the necessary

changes, copied directly from our modified version of Gadget. For reference, it takes Gadget approximately 8.5 seconds to save on disk all particle positions for a single time step, when using 4096 processors. The simulation takes on average 0.6 seconds for a single time step using 4096 processors.

Tessellation. Most our analysis uses Voronoi tessellations. We use `tess`, a tessellation code that implements the algorithm of Peterka et al. [3] In cosmology, Voronoi tessellations can be used to estimate mass density from N-body simulations, for example, as particle mass divided by its Voronoi cell volume. After normalization, we can treat the result as a probability density function and compute its information entropy using `entropy` code.

We always run `tess` and `entropy` on the same nodes, but we try two different regimes for placing the `tess`–`entropy` pair and the simulation. In the first case, we run all three in situ, on the same set of nodes, with all data transfers between codes occurring by passing pointers using Henson data facilities. Because the `tess`–`entropy` analysis of a single time step takes a lot longer than the simulation, and because we don't need to analyze every time step, we try a *fair* scheduling regime. An auxiliary puppet times how long `tess`–`entropy` took to analyze the last time step. It then blocks analysis until the simulation has run for at least as long. At this point it transfers control back to analysis, timing its performance.

The second regime is in transit. Simulation and `tess`–`entropy` analysis are placed into two separate execution groups, as shown in Listing 4. In this case, the total number of processors is split evenly among them, but one can adjust the split with a command-line parameter to `henson`. The simulation and analysis communicate asynchronously using `send` and `receive` tools: the analysis requests the data from the simulation only when it's ready to analyze it.

Light cone. We also use a particle filtering code, `lightcone`, which constructs the cosmic structure as it would be observed at a fixed space–time location. Light cones are traditionally produced by saving enough time steps from a simulation and then stacking them in redshift shells. Doing this in situ is both more computationally efficient and more accurate, since the resulting light cone has shells as fine as the simulation time steps.

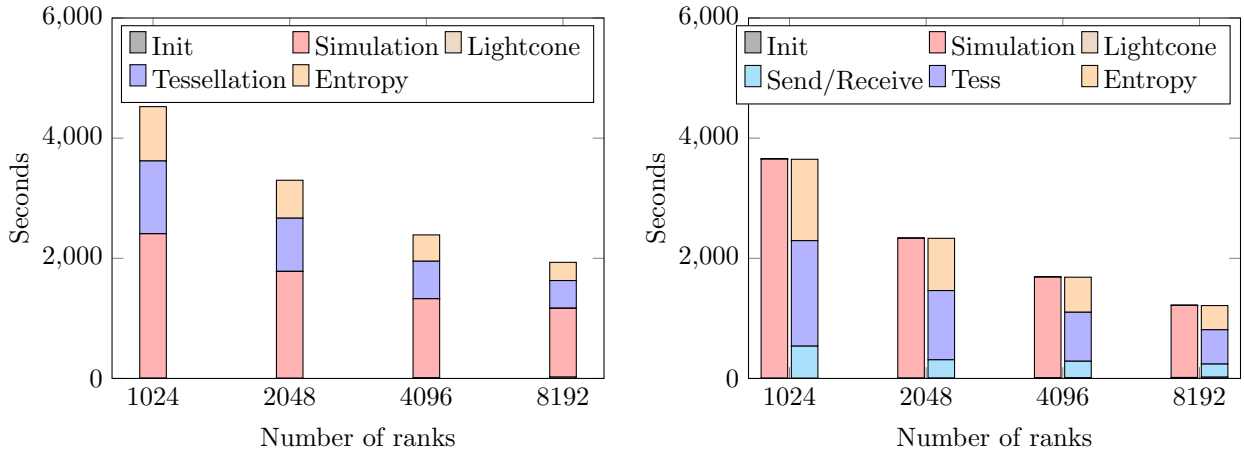
The analysis is simple: it requires a single pass over the data and accumulates those particles that match a particular criterion, saving them to disk once the simulation ends its run. We run `lightcone` in situ with the simulation because it requires little memory or time overhead. In this regime, less than a second is spent on the analysis during an hour-long simulation run.

Evaluation. Figure 1a shows the results of the (fair) in situ runs, where Gadget simulates 2001 time steps. We note that (1) Henson initialization is negligible, taking less than 25 seconds on 8192 cores; (2) the time spent in analysis (`tess` and `entropy`) never exceeds the time in simulation, balance achieved by the fair scheduling; and (3) more and more time steps are analyzed in the time it takes to simulate 2001 time steps, from 213 steps using 1024 cores to 353 steps using 8192 cores — analysis scales better than the simulation.

Figure 1b shows the results of in transit runs, where half of the processors are dedicated to the simulation, and the other half to the analysis. Again, Gadget simulates 2001 time steps. The first thing to note is that communication on the simulation side (`send` time) is negligible: from 6.97 seconds for 1024 cores down to 1.60 seconds for 8192 cores. In other words, in this regime Gadget has virtually no overhead; the processors dedicated to it spend almost all of their time simulating the universe. This makes scheduling predictable.

On the other hand, communication on the analysis side (`receive` time) is significant: when analysis finishes, it returns control to `receive`, which signals to the `send` that it is ready for more data. Such a request is processed once the simulation completes its current time step and passes control to `send`. The wait times accumulate over the course of the simulation.

It may seem that in situ analysis is performing better since more time steps get analyzed. But it also takes longer. To get a meaningful comparison of the two, we estimate how long the simulation and analysis would run in situ if they were to analyze the same number of time steps as in transit (using the same total number of cores). Specifically, we calculate $s + (t + e) \cdot (i_2/i_1)$, where s, t, and e are the in situ simulation, tessellation, and entropy times, and i_1 and i_2 are the number of steps analyzed in situ and in transit, respectively. We compare the results to the total running time of the in transit simulation (and, by construction, analysis); see Figure 2.



(a) In situ **(b)** In transit

Figure 1: Gadget–tess–entropy pipeline, in situ vs. in transit. Init, Lightcone, and Send times are so small that they are imperceptible in the plot.

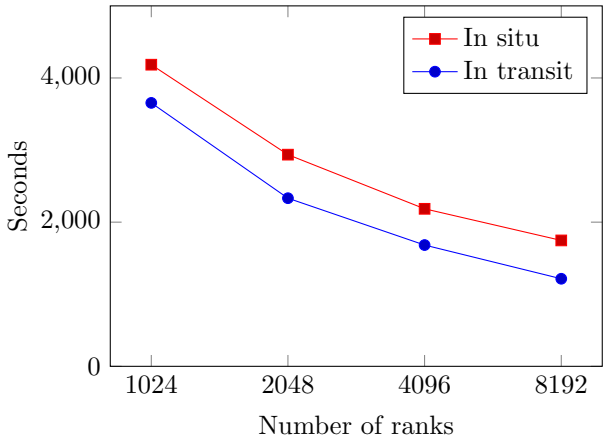


Figure 2: Simulation and analysis scaling comparison in situ vs. in transit. In situ analysis time is normalized to match the number of steps analyzed in transit.

As the figure illustrates, despite the communication overhead, in transit analysis performs better. Although surprising at first, there is a simple explanation: neither simulation, nor analysis scale perfectly. When running in situ, both have to use twice as many processors as they would in transit. The higher overheads slow the execution. But in other examples, like `lightcone`, in situ is more efficient. This highlights the importance of a flexible system, like the one presented, that seamlessly supports different execution regimes.

Acknowledgements

We would like to thank Matthew Wolf and Patrick O'Leary for helpful discussions, and Wes Bethel for his support. We are grateful to our colleagues, Wes Bethel, Peter Nugent, Tom Peterka, and Rollin Thomas, for providing feedback on draft versions of this paper. Calculations presented in this paper used resources of the National Energy Research Scientific Computing Center (NERSC). Both NERSC and the authors were supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. ZL was in part supported by the SciDAC program funded by the U.S. Department of Energy.

Disclaimer

References

[1] D.E. Knuth. *Fundamental Algorithms. The Art of Computer Programming 1.* Addison–Wesley, 1997.

[2] V. Springel. The Cosmological Simulation Code GADGET-2. *Monthly Notices of the Royal Astronomical Society*, 364:1105–1134, 2005.

[3] T. Peterka, D. Morozov, and C. Phillips. High-Performance Computation of Distributed-Memory Parallel 3D Voronoi and Delaunay Tessellation. In *Proceedings of the SC*, pages 997–1007, 2014.